Improving mobile app development using transpilers with maintainable outputs

Vinícius Jorge Vendramini, Alfredo Goldman, Grégory Mounié

Improving mobile app development using transpilers with maintainable outputs

Vinícius Jorge Vendramini
vinivendra@gmail.com
Universidade de São Paulo
São Paulo, Brazil

Alfredo Goldman
gold@ime.usp.br
Universidade de São Paulo
São Paulo, Brazil

Grégory Mounié
Gregory.Mounie@imag.fr
Université Grenoble-Alpes
Grenoble, France

ABSTRACT

Mobile application developers often target both iOS and Android in an effort to extend their target user base. There are several tools that can aid this development effort, allowing developers to maintain a single codebase for both platforms instead of two. These tools often face a few shortcomings, of which two are noteworthy: they are hard-to-replace dependencies in the codebase, and often present some type of obstacle for integrating platform-independent native code with the shared codebase. The goal of this work is to propose a new approach to creating cross-platform development tools that improves on these two aspects, and to analyze the viability of a real-world implementation of the proposed approach. An analysis of the current state of the practice indicates cross-platform compilers as a promising direction, and a study is made on the common concerns and challenges faced when developing these compilers. Based on these analyses, this work proposes the creation of a compiler that translates one platform's hand-written native code into maintainable native code for the other platform. The feasibility of implementating this approach is tested with the development of a proof-of-concept compiler from Swift to Kotlin. An analysis is made on the readability of the resulting prototype's output code, as well as other relevant metrics. The conclusion is that, while some trade-offs might be necessary, such an approach is viable if applied in an adequate ecosystem.

CCS CONCEPTS

• **Software and its engineering → Source code generation**; • **General and reference → Cross-computing tools and techniques**.

KEYWORDS

transpilers, transcompilers, mobile, Swift, Kotlin

ACM Reference Format:
Vinícius Jorge Vendramini, Alfredo Goldman, and Grégory Mounié. 2020. Improving mobile app development using transpilers with maintainable outputs. In *34th Brazilian Symposium on Software Engineering (SBES '20), October 21–23, 2020, Natal, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3422392.3422426

1 INTRODUCTION

When developing applications (apps) in any environment, it is common for programmers to target more than one operating system in an effort to expand their potential user base. In the case of smartphones, this usually means offering an app for both Android and iOS. These operating systems can be targeted directly with each platform's *software development kit* (SDK), which is available only in specific "native" languages: Objective-C and Swift for the iOS SDK, and Java and Kotlin for the Android SDK.

The lack of a common language with direct access to both platforms' SDKs means developers cannot trivially write a single codebase that compiles to both platforms. To ease that task, several authors have created *cross-platform development tools* that run the same code on both platforms using different approaches. Early versions of these tools created websites stylized look and behave like native applications, "capitaliz[ing] on the good browser support of mobile platforms"[17, p. 122] to make their content portable. Some more modern iterations like ReactNative[10] and Flutter[13] achieve portability by implementing their own runtime frameworks that support new languages like JavaScript and Dart on both platforms. Other tools can compile one of Android's native languages into Objective-C (like J2ObjC[12]) or LLVM (like Kotlin/Native[21]), making it available for iOS apps.

All of these approaches have in common the fact that they require developers to write their programs on top of layers of abstraction (such as tool-specific languages and libraries) on which the developer's code depends. This dependency means the code will not work if the layers of abstraction are removed, and instead will have to undergo some sort of migration (which is, in its worst case, a complete rewrite of the application). Additionally, these layers create a gap between shared code and platform-specific code, which can make it difficult for one to access the other.

These problems are not only theoretical – they have been documented in both academic research and industry reports (section 2). Some approaches, however, seem promising for mitigating them. J2ObjC's ability to turn Java code into Objective-C code, for instance, may indicate a solution to the dependency issue. Since the input code can be directly compiled for Android and the output code for iOS, a developer might theoretically be able to maintain them without depending on the tool or having to migrate their code. The main impediment for this is the difficulty in reading and understanding J2ObjC's generated Objective-C code in order to maintain it, as will be shown (section 4.3).

Similarly, KotlinNative's ability to compile Kotlin code into iOS libraries may indicate a solution to the issue of integrating native and platform-specific code. The tool's generated iOS libraries offer APIs that include several usability annotations and follow idiomatic conventions, which contribute to making them easier to call from platform-specific code. However, this integration only works one way: KotlinNative's libraries cannot directly access the native code that calls them, they can only access native frameworks that have been compiled separately.

SBES '20, October 21–23, 2020, Natal, Brazil
2020. ACM ISBN 978-1-4503-8753-8/20/09.
https://doi.org/10.1145/3422392.3422426

This study proposes a new approach to creating cross-platform development tools, by improving on existing models using strategies discussed in the literature. The suggested improvements enable a development tool to both **avoid becoming a dependency** for a project and **offer seamless integration** between platform-specific and shared code.

First, this definition is constructed from an analysis of existing cross-platform development tools, and the strategies to achieve it are selected from related practical and academic works (section 2). Next, the methodology is presented – specifically, how the selected strategies were used to directly pursue the goals mentioned above (section 3). Finally, an implementation is presented as a proof of concept, and there is a discussion on how the strategies were applied in this case and on the tests performed on the implemented program (section 4).

2 RELATED WORKS

2.1 Cross-platform development tools

As the main alternative for native development, tools for the implementation of web apps, hybrid apps, and other non-native approaches account for numerous practical and academic works in mobile development.

In practice, several tools enable developers to create cross-platform apps by developing their code in one of several non-native languages. For instance, PhoneGap allows users to create "hybrid applications built with HTML, CSS, and JavaScript"[1]; React Native enables the creation of apps "using only JavaScript" with "the same fundamental UI building blocks as regular iOS and Android apps"[10]; and XMLVM can "cross-compile" an Android application so that it will run on iOS[33]. Similarly, the academic literature includes several studies that propose the creation of development tools using new *domain-specific languages*, such as md2[16], ScaMo[22], Xmob[11], ICPMD[8], and the Common framework[29, 30].

In a more native setting, JetBrains developed a Kotlin compiler for iOS called Kotlin/Native[21], which can create iOS libraries with reasonably idiomatic APIs for Objective-C and Swift. Kotlin/Native libraries can only access native iOS code that has itself been compiled into libraries, which limits the possibilities for native and non-native code integration. Additionally, because the Kotlin code is compiled into binary libraries, the iOS version still depends on Kotlin/Native to be maintained.

Developers at Dropbox have taken a different approach, implementing their shared logic using C++[15]. Because C++ is available on both platforms, the code can be compiled natively and is maintainable without any external dependency; however, it does not have direct access to the native SDKs (which are only available in Java, Kotlin, Objective-C and Swift), so it faces the same integration problems mentioned above. Similarly, developers at Readdle[34] use Swift code on both iOS and Android, but the Android version cannot directly access the native SDKs.

Google's J2ObjC[12] also allows developers to maintain a native codebase. By translating Java source files into Objective-C, the tool ensures Android apps do not depend on it to be maintained, as the Java codebase can be compiled directly. The iOS apps, however, are built with the translated Objective-C code (which can be "ugly" and "hard-to-read"[4]) and have to be linked against J2ObjC's implementation of the Java standard library for iOS, making them dependent on the tool. That said, J2ObjC's model seems to show the most promise in solving these dependency and integration issues, and therefore was used as a starting point for the proposal in this work.

2.2 Transpilers

Transpiler is the name given to a compiler that translates code from a language with a high level of abstraction into another. For example, while a tool that turns C++ code into machine code might be called a compiler, one that turns Java code into Objective-C (like J2ObjC above) might be called a transpiler.

Researchers have been studying transpilers at least since the 1980's, with the works of Wallis[38] and Albrecht et al.[2] using Ada. More recently, some studies have also focused on different uses for transpilation, which include performing compiler optimizations[39], translating between more than two languages with similar paradigms,[3, 35] and creating new languages that ease correctness proofs for programs[32].

Some authors, like Huijsman, van Katwijk, Pronk, and Toetenel, have studied transpilers as a way of helping developers permanently migrate a codebase from one language to another[19]. These transpilers prioritize their output code's readability over preserving the input code's semantics, a method that allowed developers to understand the translated code and fix its bugs during the migration process (but that is not as well-suited for day-to-day compilations). This model is used today by some transpilers of mobile app languages, like SwiftKotlin[25] and Kotlift[24].

Other transpilers were designed to be a part of a developer's day-to-day compilation cycle, and as such prioritize preserving the input code's semantics so that the output code does not have to be edited after translation. This is the case with J2ObjC. Schaub and Malloy notably developed a semantics-preserving transpiler between Java, C++ and Python that aimed to produce readable code[35]. Their study highlights the importance of choosing input and output languages with compatible paradigms and semantics, noting that when "source language features are used [...] that are not present in the target language, those features must be emulated [...]; depending on the difficulty of the emulation, the readability [...] of the resulting code can suffer greatly"[35, p. 6].

Schaub and Malloy also mention that "Ideally, a [...] translation should map standard library calls for the source language to the native standard library calls for the target language, rather than introducing an interoperable library layer"[35, p. 6]. This approach was used for the GOOL transpiler's translations[3], which serve as a direct inspiration for this work.

2.3 Dependency and integration

The two main problems being investigated in this work involve a tool's dependency – i.e. the fact that codebases depend on the tools in order to be maintained – and integration – i.e. the fact that codebases that use the tools do not have direct access to and from native code. These problems have been reported both by academic researchers and by developers working in well-known applications.

The dependency issue, for instance, has been documented by developers at Facebook, who had to "rebuild" their iOS app's codebase in 2012[9]; Dropbox, who in 2019 moved from their model using shared C++ code[15]; and AirBnB, who went from using ReactNative towards a fully native approach in 2018, saying they expected the process to take over a year[28].

Similarly, integration issues in tools using non-native approaches have been well documented in the academic literature[5–7, 18, 20, 23, 27]. In particular, several of these authors describe difficulty accessing some of the operating system's functionalities, "such as a camera or a GPS sensor"[17, p. 122]. More native approaches, however, have since improved on this panorama (as described above), making them the basis for this study.

3 METHODOLOGY

Based on the analyzed works, we can properly define the goals of this work as follows. We propose the creation of a transpiler that:

(1) **Prioritizes its output code's maintainability**: generated output code should be reasonably understandable, such that a developer that knows the input and output languages and understands the input code can also reasonably understand the output code.
(2) **Depends only on maintainable or removable libraries**: generated output code should depend only on libraries that are small and understandable enough to reasonably be maintained or removed by an app's developers.
(3) **Ensures its output code offers idiomatic APIs**: generated APIs should follow common platform and language conventions and be directly accessible from platform-specific code, without requiring interoperability layers.
(4) **Ensures its output code can directly access platform-specific code**: platform-specific code written in both the input and output languages should be directly accessible from the shared code.

A developer that creates an app using a transpiler like this can switch to a fully native approach trivially, since the codebases and dependencies for both platforms would be maintainable at all times. This would avoid the migration issues reported by Facebook[9] and AirBnb[28], as mentioned above. These goals would also allow platform-specific code and shared code to integrate seamlessly with each other, thus solving our two targeted issues (dependency and integration).

The success of failure of this approach, as with the creation of any transpiler, is highly dependent on the choice of input and output languages. In general, transpilers work better when their languages are more similar, with similar paradigms, memory management techniques, basic semantics, and even syntactic structure. In the case of J2ObjC, Java and Objective-C have many aspects that are alike (such as object-oriented paradigms and reference semantics), but also some that are different (such as their use of namespaces and calling conventions) that can make it more difficult to achieve the goals above.

With this in mind, it was decided that this work's attempt at implementing the proposed approach would be a transpiler from Swift to Kotlin. While these languages have some semantic differences (whose effects will be discussed later), the similarities in their paradigms and syntax made them good candidates for this work.

The proof-of-concept transpiler was itself implemented in Swift, and is called Gryphon. The strategies below illustrate ways in which Gryphon's design builds on J2ObjC's. Most of them were (either directly or indirectly) inspired by transpilers from the literature.

Strategies for improving code maintainability:

Avoiding significant structural modifications: occasionally, transpilers need to significantly modify the structure of the input code in order to guarantee the same behavior in the output code (see J2ObjC's example in section 4.3). That can lead to problems such as misplaced comments, variable names that do not make sense when out of order, etc. Structural modifications in Gryphon are limited to changes that do not modify blocks of statements, such as:

- adding declarations, like automatic implementations of Swift's `rawValue` initializers for Kotlin's enum classes;
- moving entire declarations, like placing Swift's `static` functions inside Kotlin's `companion` objects;
- or changing a declaration's interface, like turning Swift's "`var description`" into Kotlin's "`fun toString()`".

Idiomatic translations: input and output languages often have different conventions and mechanisms concerning how to execute certain tasks, how to name certain structures, etc. Some of these differences require changes because they can affect the code's behavior: in the example above, Swift's "`var description`" has to be translated as Kotlin's "`fun toString()`" because these are the interfaces that are accessed when an instance has to be turned into a string.

Other cases involve more aesthetic changes that have the same behavior but can make the output code easier to understand by conforming to community conventions. For instance:

```swift
// Swift
guard x != nil else {
  return
}
```

```kotlin
// Kotlin translation (Swift-like)
if (x != null) {
  return
}
```

```kotlin
// Kotlin translation (idiomatic)
x ?: return
```

Style: certain aspects of coding style can be applied to the output code after the translation using tools called linters (for instance, the Kotlin community has a linter called ktlint[31]). Some coding style aspects that are not enforced by a language's linter can be enforced by the transpiler. For instance, ktlint does not check for the maximum number of characters in a line of code, but Gryphon can limit this by adding newlines when translating function calls, closure declarations, etc.

Comments: source code comments are kept during the translation and placed somewhere as similar to the input code as possible. Albrecht et al.[2] perform this task with mixed results by linking comments to nearby statements so that they move along with the code they (hopefully) refer to; inspired by that, Gryphon turns comments into their own statements, which allows them to be moved cohesively or independently as needed.

Formatting: any additional formatting details should be mimicked in each case whenever possible. This includes factors like the amount of whitespace separating groups of statements or declarations, consistently labeling parameters in function calls, etc.

Names: the names of types, variables, and other declarations stay the same whenever possible. This rule takes less precedence over others (for instance, names can be capitalized or otherwise changed to become more idiomatic), but these modifications take care not to change or lose the meanings of the given names.

Strategies for improving library maintainability:

Size: any libraries necessary for the input and output codes to compile were be designed to be as small as possible, to make it easier for developers to maintain or remove them at will.

Transpiler libraries for both languages: transpilers that do not prioritize maintainability often provide a library only for the output language. This allows their users to write the input code without having to learn to use any transpiler-specific libraries, but it also limits the transpiler developers to having access to custom code only in the output language. Gryphon does not abide by this restriction: since the transpiler's libraries are designed specifically to be small and maintainable, it is considered acceptable that the transpiler's users learn to use them. This provides some leeway on the implementation that allows for a more effective the use of the other strategies, particularly when translating references to the languages' standard libraries.

Native translations: similar languages may also have similar standard libraries, which may allow references to the input language's library to be translated to references to the output language's library. This idea was implemented in the GOOL transpiler[3], which used files written in a domain-specific language to specify mappings for classes and methods. A similar strategy was used for Gryphon, with the difference that the mappings were written directly in the input language, which allows them to be defined for arbitrarily complex Swift expressions (instead of just class and method names). Similarly, the method for specifying their Kotlin translations was made more complex by allowing them to be declared both as literal strings or as more complex structures – which can give Gryphon valuable information on the translations to avoid bugs:

```
// Swift expression
_string.formIndex(before: &_index)
// Kotlin translation
"_index -= 1"

// Swift expression
_string1.replacingOccurrences(
        of: _string2, with: _string3)
// Kotlin translation
Template.call(.dot(
  "_string1", "replace"), ["_string2", "_string3"])

// Swift expression
Range<String.Index>(uncheckedBounds:
  (lower: _index1, upper: _index2))
// Kotlin translation
Template.call("IntRange", ["_index1", "_index2"])
```

Strategies for improving access in platform-specific code to shared declarations:

Readable API:. in addition to the idiomatic translations mentioned before, any declarations that are accessible to the platform-specific code (that is, all declarations that are not marked as `private`) need to offer readable and idiomatic APIs. This includes details like the translation of function labels: Swift's convention is that they should form a sentence-like function call, but Kotlin's convention is that they correspond to the parameter names used in the implementation:

```
// Swift
func addOne(to number: Int) -> Int {
  return number + 1
}
addOne(to: 10)
```

```
// Kotlin
fun addOne(number: Int): Int {
  return number + 1
}
addOne(number = 10)
```

Gryphon keeps track of all function declarations (and their parameters' internal names) in order to translate function calls correctly when they show up in the same file or in other files.

Access control: access control modifiers are translated into their Kotlin counterparts in most cases; when there is no counterpart (e.g. some uses of Swift's `fileprivate` modifier have no equivalent translation in Kotlin), Gryphon defaults to the next less restrictive option. This avoids platform-specific code accessing a declaration it should not on most cases while also avoiding compilation errors due to inaccessible declarations on the edge cases. Gryphon also omits the access control keywords whenever possible to avoid cluttering the output code.

Open and final: translations for classes and their members include `open` or `final` modifiers to avoid platform-specific code creating subclasses where it should not. These modifiers are also omitted whenever possible to avoid cluttering the output code.

Strategies for improving access in shared code to platform-specific declarations:

Mappings: any platform-specific declarations that have the same APIs on both platforms can be trivially accessed by translated code. When the APIs are different, the same mapping system

used for translating references to the Swift standard library can be used by developers to define translations from input to output APIs.

Manual translations: in cases where the shared code should behave differently in each platform, a manual translation system can use special comments to arbitrarily ignore Swift statements or insert Kotlin statements. Specifically, this system can be used to ignore calls to iOS-only APIs and insert calls to Android-only APIs:

```
// Swift
iOSOnlyFunction() // pegasus ignore
// pegasus insert: androidOnlyFunction()
let language = "Swift" // pegasus value: "Kotlin"
```

```
// Kotlin (Gryphon translation)
androidOnlyFunction()
let language = "Kotlin"
```

This system can also be used to fix limitations on Gryphon's current translation system, such as adding `override` keywords to methods that satisfy protocol requirements (the information for which is not trivially available in Swift).

4 RESULTS

As a proof-of-concept implementation for this work, Gryphon is an open source project and is available on GitHub[37]. The studies discussed in this section were performed on a closed beta version (`v0.3`). Gryphon has since been released to the public, and its updates are downloaded by dozens of users weekly (according to GitHub's traffic data for the repository).

As mentioned before, it is important to study what effects the strategies outlined in this work might have had on Gryphon's design. The studies below aim to do that by considering different metrics that are used in related works, such as the performance of the output code, the support of different features in the input language, etc.

4.1 Bootstrapping

Some early studies on Gryphon's viability involved attempts to translate parts of open-source iOS apps into corresponding parts of their Android versions. These attempts largely failed because of the difficulty in finding appropriate candidates. There were few well-known apps using Swift and Kotlin and that open-source enough of their codebases to compile and run all automated tests, and those that existed tended to have different architectures for iOS and Android (meaning the translated code would not fit with the rest of the Android application).

Instead, Gryphon's own source code proved to be a viable target for translation. It had been implemented using Swift features that were mostly already supported, meaning relatively little adaptation was required. Possibly the greatest challenge was that, because the transpiler was still being developed, it did not include some features it needed to translate itself – and every feature implemented meant more code that needed to be translated.

This process of having a compiler or transpiler work on its own code is known as bootstrapping. Once the bootstrapping process had been implemented, the automated unit, integration and acceptance tests were translated by hand into Kotlin so that they could test the translated code. This then became part of the day-to-day testing cycle: run the automated tests on the Swift codebase, translate the codebase into Kotlin, then run the same automated tests on the Kotlin version.

This way, the bootstrapping process helps check if the transpiler works as intended: it ensures that enough Swift features are supported to translate its own source code, and it ensures that the behavior of the output code is the same as that of the input code to the extent that it passes the same automated tests.

Table 1: Number of source lines of code (SLOC) in the Swift and Kotlin versions of the Gryphon codebase.

	Swift project	Kotlin project
SLOC in shared files	11 397	10 986
% of total	97.78%	98.11%
SLOC in platform-specific files	259	212
% of total	2.22%	1.89%
Total SLOC	11 656	11 198

Table 2: Number of source lines of code (SLOC) with manual translation comments in the Swift project. Includes lines that are inserted, ignored or replaced, or lines with annotations like *open* and *override*.

	Inserted	Ignored	Replaced	Annotated	Total
SLOC	74	23	12	243	352
% of total	0.65%	0.20%	0.11%	2.14%	3.09%

Table 1 provides numbers on the amount of source lines of code involved in both the translated and platform-specific source files. The codebase includes 11 397 lines of code in files that it automatically translates to Kotlin (representing 97.78% of the total) and 259 (2.22%) in files that are exclusive to the Swift version (whose Kotlin counterparts are manually maintained). The translated Kotlin version is made up of 10 986 lines of code from translated files (representing 98.11% of the Kotlin codebase) and 212 (1.89%) from Kotlin-exclusive files.

Most of the translated files in the Swift version use manual translation comments, as detailed in table 2. Of the 11 397 source lines of code in those files, 352 of them (3.09%) use these mechanisms. This includes 74 manual insertions; 23 manual deletions; and 255 comments used for different tasks, such as adding `override` annotations to satisfy protocol conformances, etc. Further development efforts have since decreased these numbers by automatically handling some of these cases.

These line counts were done by removing all lines that were empty or contained only comments, as well as 9 lines of Kotlin code that were used only for calling the tests. It is worth noting that some translations to Kotlin result in more lines of code than their Swift counterparts, while the contrary is less common. The slight reduction on the number of lines of code in the Kotlin version (table

Table 3: Frequency of statement types in Gryphon's codebase.

Amount	Statement name
2698	Variable Declaration
2168	Expression Statement
1795	Comment Statement
1378	Return Statement
1120	If Statement
831	Assignment Statement
697	Function Declaration
143	Class Declaration
107	For Each Statement
79	Initializer Declaration
40	Continue Statement
31	Throw Statement
30	While Statement
27	Break Statement
23	Struct Declaration
20	Do Statement
20	Catch Statement
18	Companion Object
15	Switch Statement
12	Defer Statement
12	Import Declaration
5	Enum Declaration
1	Protocol Declaration

Table 4: Frequency of expression types in Gryphon's codebase.

Amount	Expression name
17106	Declaration Reference Expression
4592	Dot Expression
4268	Call Expression
3457	Tuple Expression
3436	Literal String Expression
2647	Binary Operator Expression
2022	Type Expression
1712	Template Expression
1341	Literal Int Expression
900	Nil Literal Expression
830	Tuple Shuffle Expression
687	Array Expression
452	Literal Bool Expression
339	Interpolated String Literal Expression
276	Closure Expression
235	Subscript Expression
166	Literal Code Expression
148	Prefix Unary Expression
137	Literal Character Expression
119	Optional Expression
85	Dictionary Expression
63	Force Value Expression
61	Parentheses Expression
47	If Expression

1) is likely due to other factors such as differences in code styles, in the amounts of newlines, etc.

Gryphon's required libraries are comprised of one Swift file and one Kotlin file, to be added to the iOS and Android apps respectively. The Swift file contains 649 lines of code, including 227 lines implementing the 105 standard library translations, and 422 implementing wrappers for collections. With regard to maintainability, the 227 lines implementing translations have no behavior and can be deleted safely; and the 422 lines of wrappers include only one-line methods[1] that redirect to calls to native implementations.

The Kotlin file focuses on providing Kotlin implementations of some Swift algorithms that were deemed too complex or too verbose to replicate inline without the aid of additional implementations. It contains 60 lines of code divided into seven methods. Three of these methods, like their Swift counterparts, contain exactly one line of code; three of them comprise an implementation of the *quicksort* algorithm; and one searches for a character in a string.

Table 3 and table 4 show how many statements, declarations and expressions of each kind are currently present in Gryphon's source code. They give a sense of what kinds of Swift features are currently supported and how often they appear. Some of them have noteworthy details or explanations, described below.

Expression Statement: corresponds to an expression that acts as a standalone statement - e.g. calling a function like `print("Hello!")`.

[1]There are three methods with 5 lines that are exceptions to this rule. These methods are used to implement covariant casts for the data structures: they check if the casted element types are compatible subtypes, and return either the successfully casted type or nil accordingly.

Comment Statement: a "fake" statement created to contain a source comment.

If Statement: encompasses several variations of an `if` statement in Swift, including common `if`s and `else`s; `if let`s and `if case let`s; and `guard`s. In particular, `if let`s may be translated as multiple statements in Kotlin, given the need to declare the variables before testing them.

```
// Swift
if let userName = networkResult {
  // ...
}
```

```
// Kotlin (Gryphon translation)
val userName: String? = networkResult
if (userName != null) {
  // ...
}
```

Throw, Do and Catch Statements: exception handling in Swift is performed in a similar manner to Kotlin. Notably, Kotlin is more permissive in its syntax and does not require users to annotate functions declarations with `throws` or individual calls to these functions with `try`. This means that many of Swift's exception annotations can be omitted in Kotlin while maintaining the same semantics.

```swift
// Swift
func dangerousFunction() throws {
  throw MyError()
}

do {
  try dangerousFunction()
}
catch let error {
  print(error)
}
```

```kotlin
// Kotlin (Gryphon translation)
internal fun dangerousFunction() {
  throw MyError()
}

try {
  dangerousFunction()
}
catch (error: Exception) {
  println(top-level)
}
```

Enum and Struct Declarations: Swift's `structs` are translated as Kotlin's data classes; common `enums` are translated as `enum classes`; and `enums` with associated values are translated as `sealed classes`. Each of these translations has similar meanings and behaviors, but they differ in their semantics – while Swift's types are passed by value, Kotlin's types are passed by reference. This difference can cause differences in behavior when the types are mutable, which can lead to bugs. Gryphon tries to avoid these bugs raising warnings when developers declare mutable members on `structs` and `enums`. The warnings can also be silenced, allowing developers to proceed once they are aware of the risks. This problem is inevitable given the semantics of the two languages, but it might be avoided with different language combinations.

Array and Dictionary expressions: Similarly, Swift's `Array` and `Dictionary` collections are passed by value, while Kotlin's corresponding `Lists` and `Maps` are passed by reference. The initial attempt to solve this problem involved recreating Swift's behavior in Kotlin: if the collections were copied every time a new reference was created, the behavior would be consistent. However, this approach could significantly hurt performance.

Swift leverages its *automatic reference counting* mechanism to implement a copy-on-write optimization: collections are only copied when they are about to me mutated *and* the language knows that more than one reference is pointing to the collection; otherwise, Swift only copies the reference to the collection. Kotlin uses a *garbage collector* instead of *automatic reference counting*, meaning it does not have the necessary information on the amount of references to make the same optimization[2]. Because of this, Gryphon takes the opposite approach: it mimics Kotlin's behavior in Swift by providing wrappers to Swift's collections that are passed by reference. These wrappers leverage Swift's metaprogramming capabilities to provide exactly the same API as the collections they wrap, making them more user friendly.

[2]Kotlin does have access to Java's *CopyOnWriteArrayList*, but that class implements a different algorithm than Swift and is meant for specific use cases, as specified in its documentation[26].

Switch Statement: Swift's `switch` statements are translated into `when` statements, which can be used as expressions in Kotlin. This enables Gryphon to translate some switch statements idiomatically – for instance, when returning a value from a function:

```swift
// Swift
func isXZero(x: Int) -> Bool {
  switch x {
  case 0: return true
  default: return false
  }
}
```

```kotlin
// Kotlin (Gryphon translation)
internal fun isXZero(x: Int): Boolean {
  return when (x) {
    0 -> true
    else -> false
  }
}
```

Declaration Reference Expression: Roughly corresponds to when a variable or instance of any type is referenced, hence its high frequency in table 4.

Closure Expression: Swift's closures are translated into *lambdas* in Kotlin. This includes the translation of anonymous parameters when possible (i.e. Swift's `$0` to Kotlin's `it`). One of the challenges with this translation involves returning from closures: while `return` statements exit the closure in Swift, they exit the innermost function declaration in Kotlin. To get the same behavior, Gryphon currently scans the function that is calling the lambda and adds explicit labels to any return statements it finds.

```swift
// Swift
[1, 2, 3].map { return $0 + 1 }
```

```kotlin
// Kotlin (Gryphon translation)
listOf(1, 2, 3).map { return@map it + 1 }
```

Tuple Expression: Swift supports tuples as native types, functioning similarly to anonymous `structs`; Kotlin does not support tuples natively, but it does offer a `Pair` type in its standard library that serves as a translation for tuples with two elements. Tuples are also used to list parameters in most function calls.

```swift
// Swift
let tuple: (Int, Int) = (1, 2)
print(tuple.0)
print(tuple.1)
```

```kotlin
// Kotlin (Gryphon translation)
val tuple: Pair<Int, Int> = Pair<Int, Int>(1, 2)
println(tuple.first)
println(tuple.second)
```

Tuple Shuffle Expression: appear when on function calls that include parameters with default values (which can be omitted) or variadic parameters (which can receive multiple values). In particular, the Kotlin translation has to omit labels before variadic parameters on function calls.

```swift
// Swift
func tupleShuffle(a: Int, b: Int..., c: Int = 0) {
}

tupleShuffle(a: 1, b: 1, 2, 3)
tupleShuffle(a: 1, b: 1, 2, 3, c: 1)
```

```kotlin
// Kotlin (Gryphon translation)
fun tupleShuffle(a: Int, vararg b: Int, c: Int = 0) {
}

tupleShuffle(1, 1, 2, 3)
tupleShuffle(1, 1, 2, 3, c = 1)
```

Literal Code Expression: an expression corresponding to code that has been literally translated or inserted into the output. This means both code that a developer inserted manually and translations that involve literal code, such as references to the Kotlin standard library.

Optional Expression: Optional expressions specify when a value with an optional type (i.e. a type that can either be present or be `null`) is being evaluated. Their translation is particularly complicated in optional evaluation chains, which have to be propagated from Swift to Kotlin.

If Expression: If statements in Kotlin can be used as expressions; this is particularly useful in this case because it allows Swift's ternary operator to be translated (since Kotlin itself does not support a similar operator).

```swift
// Swift
let language = isIniOS ? "Swift" : "Kotlin"
```

```kotlin
// Kotlin (Gryphon translation)
val language: String = if (isIniOS) { "Swift" } else
↪ { "Kotlin" }
```

4.2 Benchmarks

In order to ensure the proof-of-concept transpiler could be used to generate output code with reasonably acceptable performance, a series of benchmark tests were used. The benchmarks were taken from the Computer Languages Benchmark Game's catalog, which has been used previously in other research[14]. It provides a set of toy programs implemented in different programming languages, which notably include Swift and Java.

The purpose of this study was to analyze the performance characteristics of code that had not been specifically optimized (as performance-critical code can be implemented in platform-specific files). Therefore, heavily-optimized examples were discarded, leaving five programs: *binary-trees*, *fannkuch-redux*, *mandelbrot*, *n-body*, and *spectral-norm*. Four versions were created for each program: hand-written Swift and Kotlin, and Gryphon-compatible Swift and its Kotlin translation. Comparing the hand-written versions with the Gryphon versions enabled an analysis of Gryphon's impact on performance.

Table 5: Swift benchmarks run times. Times are in seconds, as an average of 600 executions. Slowdowns represent how much slower (or faster, if negative) the Gryphon version is.

Benchmark	Hand-written	Gryphon	Slowdown
Binary trees	3.55	3.53	0.56%
Fannkuch	9.18	9.14	0.43%
Mandelbrot	3.77	3.78	-0.39%
NBody	3.99	3.99	-0.07%
Spectral norm	2.88	2.87	0.55%

Table 6: Kotlin benchmarks run times, analogous to table 5.

Benchmark	Hand-written	Gryphon	Slowdown
Binary trees	3.94	3.94	-0.14%
Fannkuch	2.30	2.27	1.27%
Mandelbrot	4.38	4.36	0.24%
NBody	5.76	5.65	1.94%
Spectral norm	7.85	7.62	3.01%

The benchmarks were executed 600 times for each version of each program. All of the data from the experiments, as well as the source codes for the benchmarks, are publicly available[36].

The results are summarized on tables 5 and 6. Notably, times for the Gryphon-compatible Swift version ranged from a 0.39% speedup to a 0.56% slowdown, and times for the Gryphon-translated Kotlin version ranged from a 0.14% speedup to a 3.01% slowdown. These numbers are interpreted here as an indication that if any performance regressions did happen they are likely to be acceptable to most users.

4.3 Maintainability

Formally testing the maintainability of the proof-of-concept's output would involve testing how well production code for realistic applications can be modified and evolved by developers over time, which is outside of the scope of this work. Rather, the focus here is on the code's perceived readability, as well as on numerical analyses related to both its size and how much of it can be translated.

An early attempt at determining readability involved an online survey, which asked participants to rate the readability of the same algorithm implemented in Swift, Kotlin, Java and Objective-C, including versions that used Gryphon. This survey was not answered by enough developers for a statistically significant result, and so it is not included in this text, but the results are available in the accompanying material. In particular, comments on the survey suggested developers' main concerns were the chosen language and the algorithm's documentation.

Following these experiments, a handful of the authors' colleagues and peers suggested that Gryphon's readability improvements would be self-evident in comparison with a transpiler that did not prioritize readability. As such, the code snippets below compare a translation using Gryphon with one using J2ObjC (which served as

the main basis for Gryphon's development). It is worth remarking that the intent here is not to criticize J2ObjC, which has been a valuable part of the mobile development ecosystem, but to highlight a specific contribution made in this work.

Consider the definition of an empty class called `Hello`. Xcode, the main development environment for iOS projects, creates two Objective-C files, one for the implementation and one for the interface (lightly edited below).

```
// Objective-C interface (Xcode template)

NS_ASSUME_NONNULL_BEGIN
@interface Hello : NSObject
@end
NS_ASSUME_NONNULL_END
```

```
// Objective-C implementation (Xcode template)

#import "Hello.h"
@implementation Hello
@end
```

Similarly, we could define an empty `Hello` class in Java as follows:

```
// Java

class Hello { }
```

J2ObjC translates this Java class into an Objective-C implementation file and an interface file. The interface file contains 21 source lines of code, and the implementation file contains 33. Below are some snippets of this translation (edited for brevity):

```
// Objective-C interface (J2ObjC translation)

@interface Hello : NSObject
#pragma mark Package-Private
- (instancetype)init;
@end

J2OBJC_EMPTY_STATIC_INIT(Hello)
FOUNDATION_EXPORT void Hello_init(Hello *self);
FOUNDATION_EXPORT Hello *new_Hello_init(void)
  NS_RETURNS_RETAINED;
FOUNDATION_EXPORT Hello *create_Hello_init(void);
J2OBJC_TYPE_LITERAL_HEADER(Hello)
```

```
// Objective-C implementation (J2ObjC translation)

@implementation Hello

J2OBJC_IGNORE_DESIGNATED_BEGIN
- (instancetype)init {
  Hello_init(self);
  return self;
}
J2OBJC_IGNORE_DESIGNATED_END

+ (const J2ObjcClassInfo *)__metadata {
  static J2ObjcMethodInfo methods[] = {
    { NULL, NULL, 0x0, -1, -1, -1, -1, -1, -1 },
  };
  #pragma clang diagnostic push
  #pragma clang diagnostic ignored
    "-Wobjc-multiple-method-names"
  #pragma clang diagnostic ignored
    "-Wundeclared-selector"
  methods[0].selector = @selector(init);
  #pragma clang diagnostic pop
  static const J2ObjcClassInfo _Hello =
    { "Hello", NULL, NULL, methods, NULL, 7, 0x0,
      1, 0, -1, -1, -1, -1, -1 };
  return &_Hello;
}

@end

void Hello_init(Hello *self) {
  NSObject_init(self);
}

Hello *new_Hello_init() {
  J2OBJC_NEW_IMPL(Hello, init)
}

Hello *create_Hello_init() {
  J2OBJC_CREATE_IMPL(Hello, init)
}

J2OBJC_CLASS_TYPE_LITERAL_SOURCE(Hello)
```

In Gryphon's case, the input and output code for an empty class would be as follows:

```
// Swift
class Hello { }
```

```
// Kotlin (Gryphon translation)
internal open class Hello {
}
```

Similarly, J2ObjC requires users to link their iOS apps against a reimplementation of Java's standard library in Objective-C; comparatively, Gryphon requires two files that together total just over 700 source lines of code.

5 CONCLUSIONS

This work proposed an approach to creating a mobile development tool that avoids the dependency and integration issues as detailed. The proposed approach involves creating a transpiler that produces readable output code, depends on maintainable libraries, and provides seamless integration between shared code and native code.

The viability of implementing this approach was tested with a proof-of-concept transpiler, which was then compared with J2ObjC

as the next-best alternative for solving these issues. The relative simplicity and readability that were demonstrated indicate that programs that use Gryphon can depend relatively little on the transpiler: the output code is reasonably readable and concise, and the transpiler's libraries are reasonably maintainable or removable. Additionally, the transpiler's generated APIs and its manual translation mechanisms allow for seamless integration with native code. Finally, quantitative tests indicate that pursuing these goals did not cause significant regressions in other aspects of the translation process, like the generated code's performance.

Recommended directions for further works involve other implementations of this approach (especially using other language combinations that might fare better in some semantic aspects) and testing this approach in more realistic situations, particularly involving real mobile apps (so that there may be a better understanding of its limitations).

ACKNOWLEDGMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001

REFERENCES

[1] Adobe. [n.d.]. *Adobe PhoneGap.* Retrieved on 2020-06-15, from https://phonegap.com.

[2] P.F. ALBRECHT, P.E. GARRISON, S.L. GRAHAM, R.H. HYERLE, P. IP, and B. KRIEG-BRÜCKNER. 1980. Source-to-source Translation: Ada to Pascal and Pascal to Ada. In *Proceedings of the ACM-SIGPLAN Symposium on Ada Programming Language* (Boston, Massachusetts) *(SIGPLAN '80)*. ACM, New York, NY, USA, 183–193. https://doi.org/10.1145/948632.948658

[3] P. ARRIGHI, J. GIRARD, M. LEZAMA, and K. MAZET. 2014. The GOOL System: A Lightweight Object-oriented Programming Language Translator. In *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE* (Uppsala, Sweden) *(ICOOOLPS '14)*. ACM, New York, NY, USA, Article 5, 7 pages. https://doi.org/10.1145/2633301.2633306

[4] T. BALL. 2013. *cfront: a 'J2ObjC Inspiration.* Retrieved on 2020-06-15, from http://j2objc.blogspot.com/2013/09/cfront-j2objc-inspiration-developers.html.

[5] N. BARTH. 2014. *Análise Comparativa de Ferramentas de Desenvolvimento de Aplicativos Móveis Multiplataforma.* Bachelor's thesis. Universidade Regional de Blumenau.

[6] A. CHARLAND and B. LEROUX. 2011. Mobile Application Development: Web vs. Native. *Queue* 9, 4, Article 20 (April 2011), 9 pages. https://doi.org/10.1145/1966989.1968203

[7] L. CORRAL, A. SILLITTI, and G. SUCCI. 2012. Mobile Multiplatform Development: An Experiment for Performance Analysis. *Procedia Computer Science* 10 (2012), 736 – 743. https://doi.org/10.1016/j.procs.2012.06.094

[8] W.S. EL-KASSAS, B.A. ABDULLAH, A.H. YOUSEF, and A. WAHBA. 2014. ICPMD: Integrated cross-platform mobile development solution. In *2014 9th International Conference on Computer Engineering Systems (ICCES)*. 307–317. https://doi.org/10.1109/ICCES.2014.7030977

[9] Facebook. 2012. *Under the hood: Rebuilding Facebook for iOS.* Retrieved on 2020-06-15, from https://www.facebook.com/notes/facebook-engineering/under-the-hood-rebuilding-facebook-for-ios/10151036091753920.

[10] Facebook. 2020. *React Native.* Retrieved on 2020-06-15, from https://facebook.github.io/react-native/.

[11] O. LE GOAER and S. WALTHAM. 2013. Yet Another DSL for Cross-platforms Mobile Development. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages* (Montpellier, France) *(GlobalDSL '13)*. ACM, New York, NY, USA, 28–33. https://doi.org/10.1145/2489812.2489819

[12] Google. [n.d.]. *J2ObjC: Overview.* Retrieved on 2020-06-15, from https://developers.google.com/j2objc.

[13] Google. 2020. *Flutter.* Retrieved on 2020-06-15, from https://flutter.dev.

[14] I. GOUY. [n.d.]. *The Computer Language Benchmarks Game.* Retrieved on 2020-06-15, from https://benchmarksgame-team.pages.debian.net/benchmarksgame/sometimes-people-just-make-up-stuff.html.

[15] E. GUTHMANN. 2019. *The (not so) hidden cost of sharing code between iOS and Android.* Retrieved on 2020-06-15, from https://blogs.dropbox.com/tech/2019/08/the-not-so-hidden-cost-of-sharing-code-between-ios-and-android/.

[16] H. HEITKÖTTER, T.A. MAJCHRZAK, and H. KUCHEN. 2013. Cross-platform Model-driven Development of Mobile Applications with Md2. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing* (Coimbra, Portugal) *(SAC '13)*. ACM, New York, NY, USA, 526–533. https://doi.org/10.1145/2480362.2480464

[17] H.HEITKÖTTER, S. HANSCHKE, and T.A. MAJCHRZAK. 2013. Evaluating Cross-Platform Development Approaches for Mobile Applications. In *Web Information Systems and Technologies*. Springer Berlin Heidelberg, Berlin, Heidelberg, 120–138.

[18] A. HOLZINGER, P. TREITLER, and W. SLANY. 2012. Making Apps Useable on Multiple Different Mobile Platforms: On Interoperability for Business Application Development on Smartphones. In *Multidisciplinary Research and Practice for Information Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 176–189.

[19] R.D. HUIJSMAN, K.J. VAN KATWIJK, C. PRONK, and W.J. TOETENEL. 1987. Translating Algol 60 Programs into Ada. *Ada Lett.* VII, 5 (September 1987), 42–50. https://doi.org/10.1145/36077.36080

[20] S. JIANG. 2016. *Comparison of Native, Cross-Platform and Hyper Mobile Development Tools Approaches for iOS and Android Mobile Applications.* Bachelor's thesis. Department of Computer Science and Engineering, University of Gothenburg.

[21] Kotlin. [n.d.]. *Kotlin/Native.* Retrieved on 2020-06-15, from https://kotlinlang.org/docs/reference/native-overview.html.

[22] D. MACOS and A. SOLYMOSI. 2013. ScaMo: Realisation of an OO-functional DSL for cross platform mobile applications development. *AIP Conference Proceedings* 1558, 1 (2013), 327–331. https://doi.org/10.1063/1.4825490 arXiv:https://aip.scitation.org/doi/pdf/10.1063/1.4825490

[23] R. MADAUDO and P. SCANDURRA. 2013. Native versus Cross-platform frameworks for mobile application development. In *VIII Workshop of the Italian Eclipse Community*.

[24] Moshbit. 2020. *Kotlift.* Retrieved on 2020-06-15, from https://github.com/moshbit/Kotlift.

[25] A.G. OLLOQUI. 2020. *SwiftKotlin.* Retrieved on 2020-06-15, from https://github.com/angelolloqui/SwiftKotlin.

[26] Oracle. 2019. *Java SE 12 & JDK 12 Documentation: CopyOnWriteArrayList.* Available on https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/concurrent/CopyOnWriteArrayList.html, retrieved on 2020-06-15.

[27] M. PALMIERI, I. SINGH, and A. CICCHETTI. 2012. Comparison of cross-platform mobile development tools. In *2012 16th International Conference on Intelligence in Next Generation Networks*. 179–186. https://doi.org/10.1109/ICIN.2012.6376023

[28] G. PEAL. 2018. *React Native at Airbnb.* Retrieved on 2020-06-15, from https://medium.com/airbnb-engineering/react-native-at-airbnb-f95aa460be1c.

[29] J. PERCHAT, M. DESERTOT, and S. LECOMTE. 2013. Component based Framework to Create Mobile Cross-platform Applications. *Procedia Computer Science* 19 (2013), 1004 – 1011. https://doi.org/10.1016/j.procs.2013.06.140 The 4th International Conference on Ambient Systems, Networks and Technologies (ANT 2013), the 3rd International Conference on Sustainable Energy Information Technology (SEIT-2013).

[30] J. PERCHAT, M. DESERTOT, and S. LECOMTE. 2014. Common framework: A hybrid approach to integrate cross-platform components in mobile application. *Journal of Computer Science* 10 (November 2014), 2165–2181.

[31] Pinterest. 2020. *ktlint: An anti-bikeshedding Kotlin linter with built-in formatter.* Retrieved on 2020-06-15, from https://github.com/pinterest/ktlint.

[32] D.A. PLAISTED. 2013. Source-to-source translation and software engineering. *Journal of Software Engineering and Applications* 6, 04 (2013), 30.

[33] A. PUDER and O. ANTEBI. 2013. Cross-Compiling Android Applications to iOS and Windows Phone 7. *Mobile Networks and Applications* 18, 1 (February 2013), 3–21. https://doi.org/10.1007/s11036-012-0374-2

[34] Readdle. 2018. *Swift for Android: Our Experience and Tools.* Retrieved on 2020-06-15, from https://blog.readdle.com/why-we-use-swift-for-android-db449feeacaf.

[35] S. SCHAUB and B.A. MALLOY. 2016. The Design and Evaluation of an Interoperable Translation System for Object-Oriented Software Reuse. *Journal of Object Technology* 15, 4 (2016).

[36] V. VENDRAMINI. 2019. *Research data on the Gryphon transpiler.* https://doi.org/10.5281/zenodo.3489737 Retrieved on 2020-08-11, from https://github.com/vinivendra/GryphonResearch.

[37] V. VENDRAMINI. 2020. *Gryphon: The Swift to Kotlin translator.* https://doi.org/10.5281/zenodo.3489740 Retrieved on 2020-08-11, from https://github.com/vinivendra/Gryphon.

[38] P.J.L. WALLIS. 1985. Automatic Language Conversion and Its Place in the Transition to Ada. In *Proceedings of the 1985 Annual ACM SIGAda International Conference on Ada* (Paris, France) *(SIGAda '85)*. Cambridge University Press, New York, NY, USA, 275–284. https://doi.org/10.1145/324426.324399

[39] Q. YI. 2012. POET: A scripting language for applying parameterized source-to-source program transformations. *Software: Practice and Experience* 42 (June 2012). https://doi.org/10.1002/spe.1089